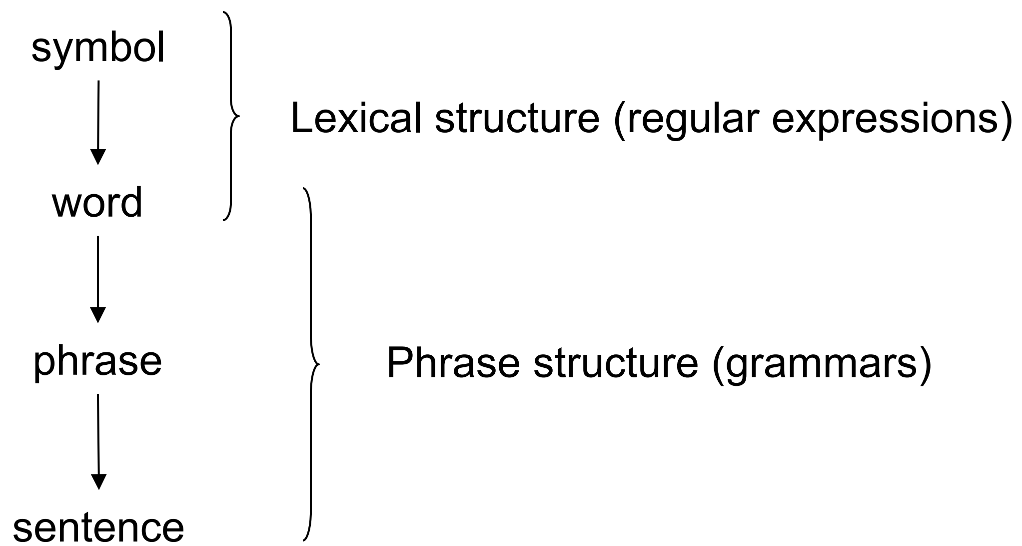
Multi-Symbol Words - Lexical Analysis

- In our exp0 programming language we only had words of length one
- However, most programming languages have words of lengths more than one
- The lexical structure of a programming language specifies how symbols are combined to form words
  - Not to be confused with the phrase structure which tells us how words are combined to form phrases and sentences
- The lexical structure of a programming language can be specified with regular expressions
  - whereas the phrase structure is specified with CFGs.
- The “parser” for the lexical structure of a programming language is called a lexical analyzer or lexer

Multi-Symbol Words - Lexical Analysis

- This gives us the following hierarchy:



Ply & Regular Expressions

- The lexer in Ply uses the Python regular expression syntax
  - https://docs.python.org/3.6/library/re.html
- Documentation on the Ply lexer can be found here:
  - http://www.dabeaz.com/ply/ply.html#ply_nn3

Regular Expressions (RE)

- REs can be defined inductively as follows:
  - Each letter 'a' through 'z' and 'A' through 'Z' constitutes a RE and matches that letter
  - Each number '0' through '9' constitutes a RE and matches that number
  - Each printable character '(', ')','+', *etc.* constitutes a RE and matches that character.
  - If A is a RE, then (A) is also a RE and matches A
    - '(A)' vs. '\(A\)'
  - If A and B are REs, then AB is also a RE and matches the concatenation of A and B.
  - If A and B are REs, then A|B is also an RE and matches A or B
  - If A is a RE, then A? is also a RE and matches zero or one instances of A
  - If A is a RE, then A* is also a RE and matches zero or more instances of A
  - If A is a RE, then A+ is also a RE and matches one or more instances of A

NOTE: Python regular expressions are written as strings, in particular as raw strings such as: `r'\(a|b\)+'`

Regular Expressions (RE)

- Useful RE Notations:
  - ‘[a – z]’ - any *single* character between ‘a’ and ‘z’
  - ‘[A – Z]’ - any *single* character between ‘A’ and ‘Z’
  - ‘[0 – 9]’ - any *single* digit between ‘0’ and ‘9’
  - . - the dot matches any character
- Also, any other character can be considered a RE. You need to distinguish between RE commands and syntax of the language to be defined:
  - i.e., ‘a+’ *vs.* ‘a\+’
- Examples
  - ‘p’ ‘r’ ‘i’ ‘n’ ‘t’ is the same as ‘print’ (why)
  - ‘-?[0-9]+’
  - ‘([a – z] | [A – Z])+[0 – 9]*’

Regular Expressions (RE)

- Exercises:
  - Write a RE for character strings that start and end with a single digit.
    - E.g. 3abc5
  - Write a RE for numbers that have at least two digits and a dot separates the first two digits
    - E.g. 3.14, 2.5, 3.0, 0.125
  - Write a RE for numbers where the dot can appear anywhere
    - E.g. 12.5, .10, 125.0, 125.678, 15.
  - Write a RE for words that start with a single capital letter followed by lowercase letters and numbers, neither of which has to appear in the word.
    - E.g. Version10a, A

The Exp1 Language

- We extend the Exp0 language to create Exp1:
  - keywords that are longer than a single character
  - Variable names that conform to the normal variable names found in other programming languages: a single alpha character followed by zero or more alpha-numerical characters
  - Numbers that consist of more than one digit.
- Ply allows you to specify both the lexer (lex) and the parser (yacc)
- It is common practice to convert words of the language longer than one character into tokens

Exp1 Lexer

Single-character words → `literals`

Multi-character words → `tokens`

```
# %load code/exp1_lex.py
# Lexer for Exp1

from ply import lex

reserved = {
  'store' : 'STORE',
  'print' : 'PRINT'
}

literals = [';','+','-','(',')']

tokens = ['NAME','NUMBER'] + list(reserved.values())

t_ignore = ' \t'

def t_NAME(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value,'NAME')    # Check for reserved words
    return t

def t_NUMBER(t):
    r'[0-9]+'
    t.value = int(t.value)
    return t

def t_NEWLINE(t):
  r'\n'
  pass

def t_error(t):
    raise SyntaxError("Illegal character {}".format(t.value[0]))

# build the lexer
lexer = lex.lex()
```

Exp1 Grammar

```
# %load code/exp1_gram.py
from ply import yacc
from exp1_lex import tokens, lexer

def p_grammar(_):
  """
  prog : stmt_list

  stmt_list : stmt stmt_list
            | empty

  stmt : PRINT exp ';'
       | STORE var exp ';'

  exp : '+' exp exp
      | '-' exp exp
      | '(' exp ')'
      | var
      | num

  var : NAME

  num : NUMBER
  """
  pass

def p_empty(p):
  'empty :'
  pass

def p_error(t):
    print("Syntax error at '%s'" % t.value)

parser = yacc.yacc()
```

Tokens

- The definition of Tokens usually has two parts:
  - A token type
  - A token value
- For example, in Exp1 we have
  - a token type PRINT with a token value of ‘print’
  - a token type NUMBER with an integer token value.

Testing the Specification

```
In [18]: from exp1_gram import parser
         from exp1_lex import lexer

Generating LALR tables

In [23]: input_stream = "store x1 10 ; print + x1 1 ;"

In [24]: parser.parse(input=input_stream, lexer=lexer)
```

Writing an Interpreter for Exp1

- Writing an interpreter for Exp1
  - We add actions to the grammar rules that interpret the values within the phrase structure of a program.
  - Observation: we need access to the token values during parsing in order to evaluate things like the values of numbers or the value of an addition.
  - Observation: interpretation always starts at the leaves.

Writing an Interpreter for Exp1

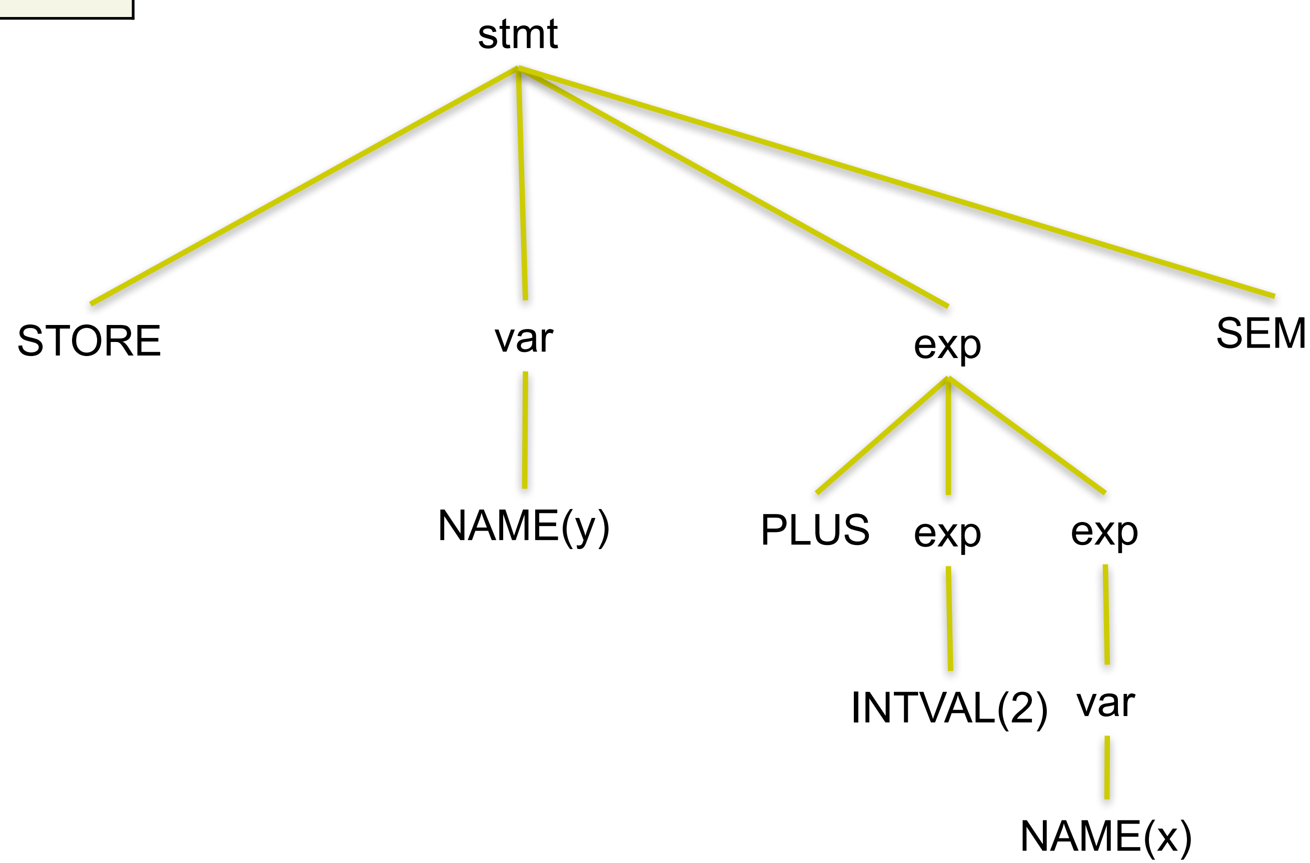
- Consider the following Exp1 program:

```
store y + 2 x ;
```

- Where x has the value 3.

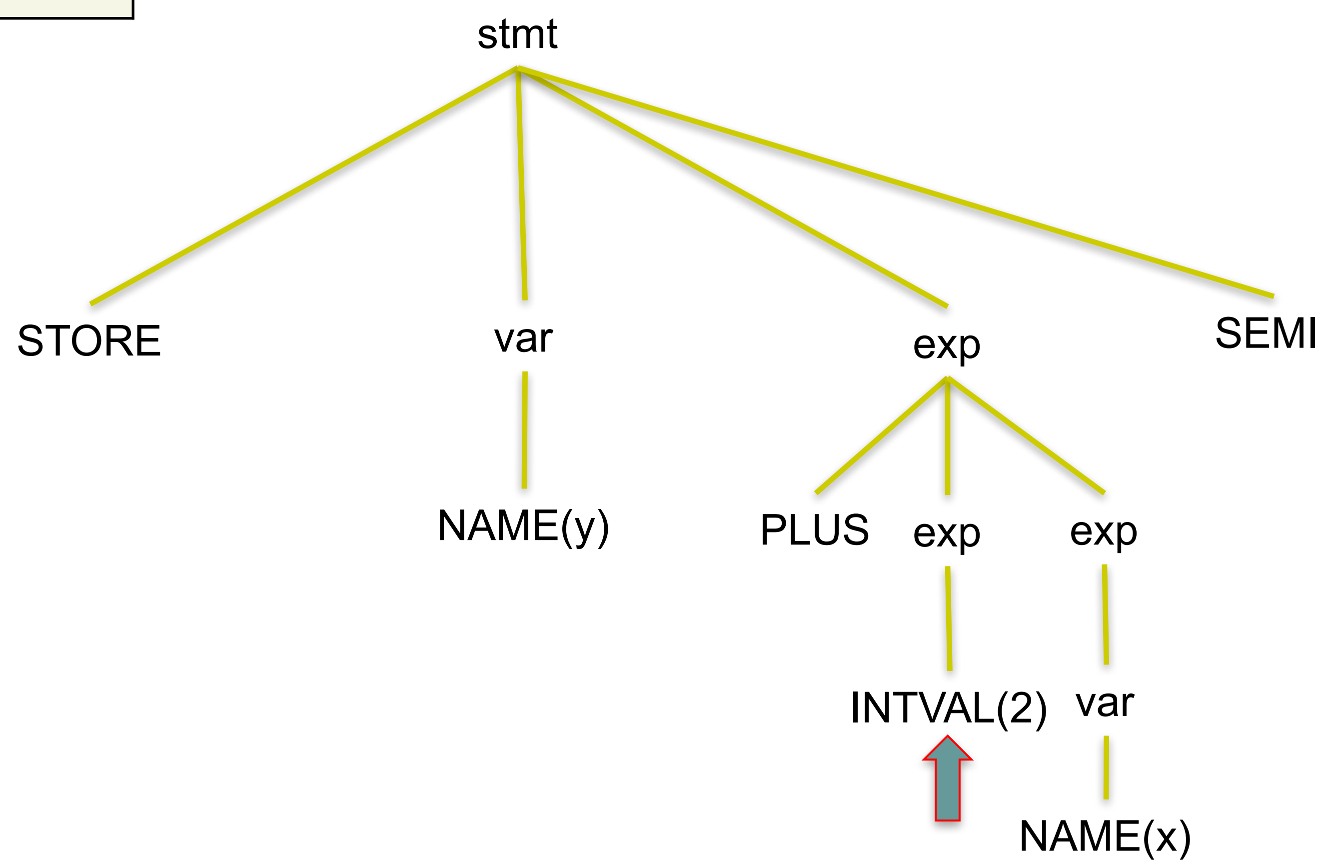
Symbol Table	
x	3
y	???

Action: start



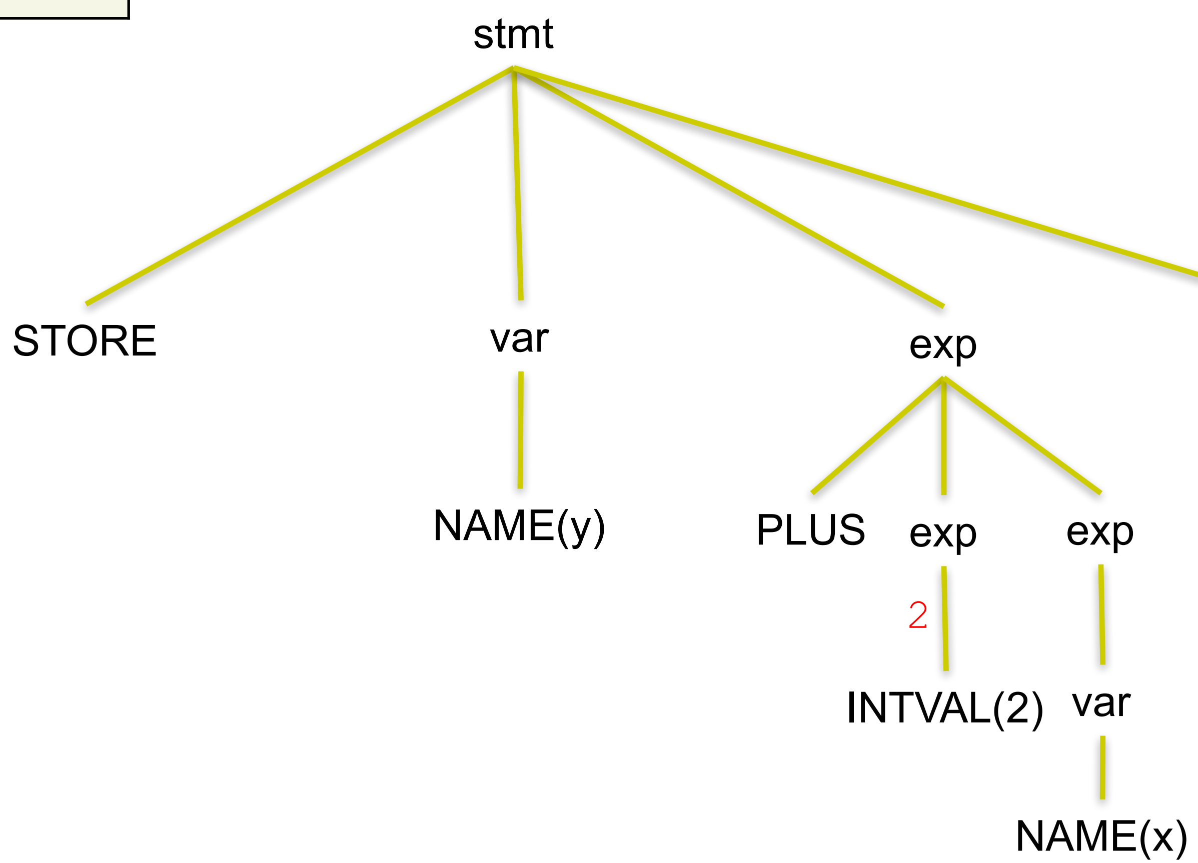
Symbol Table	
x	3
y	???

```
Action: interpret INTVAL
```



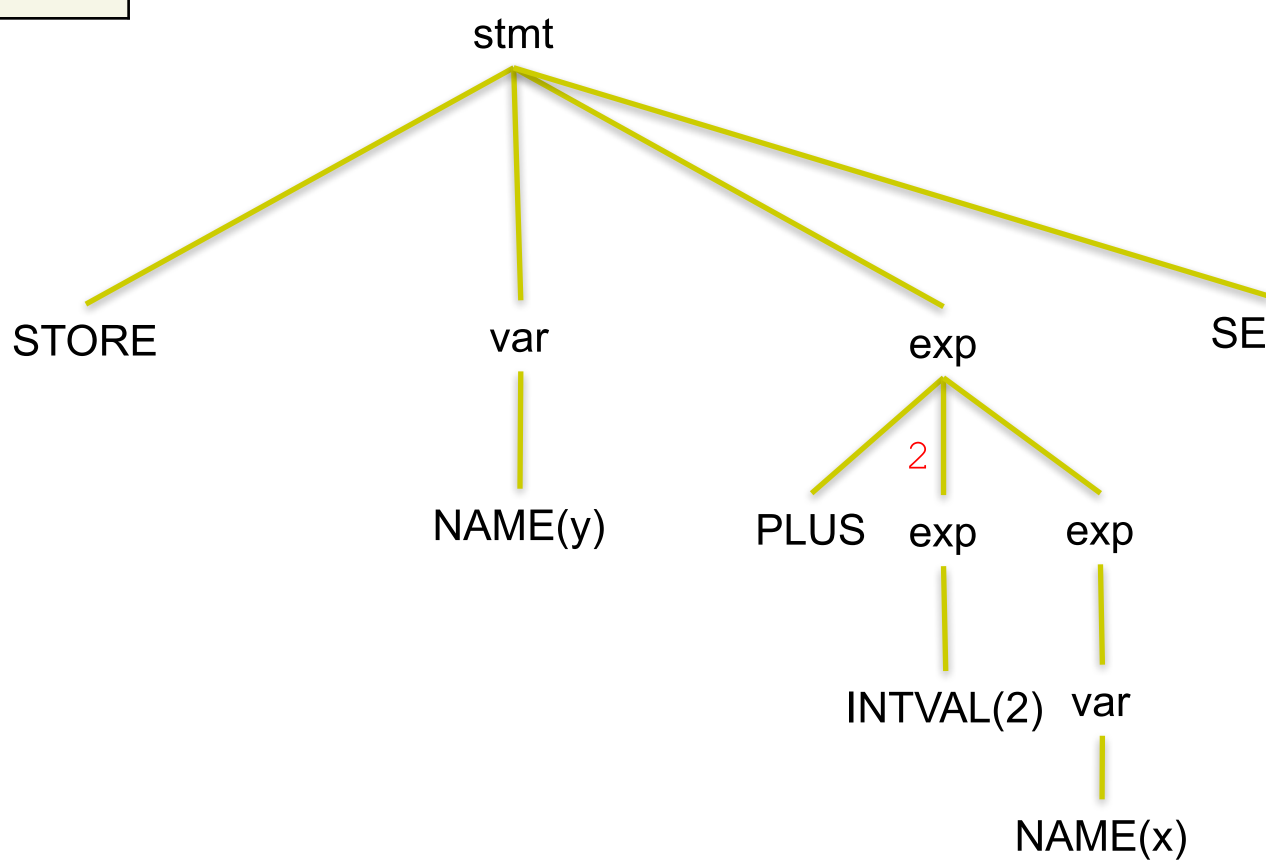
Symbol Table	
x	3
y	???

Action: propagate



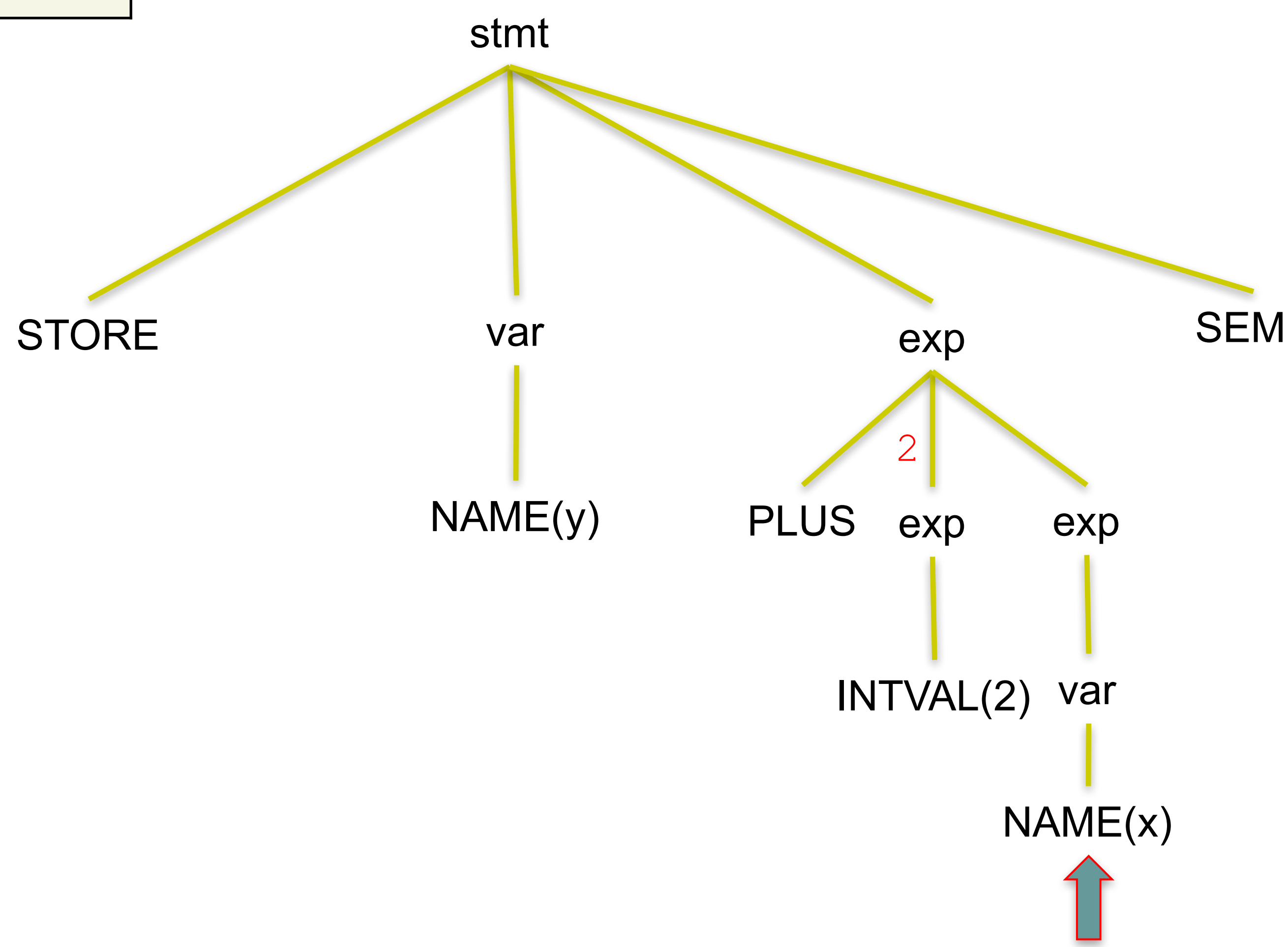
Symbol Table	
x	3
y	???

Action: propagate



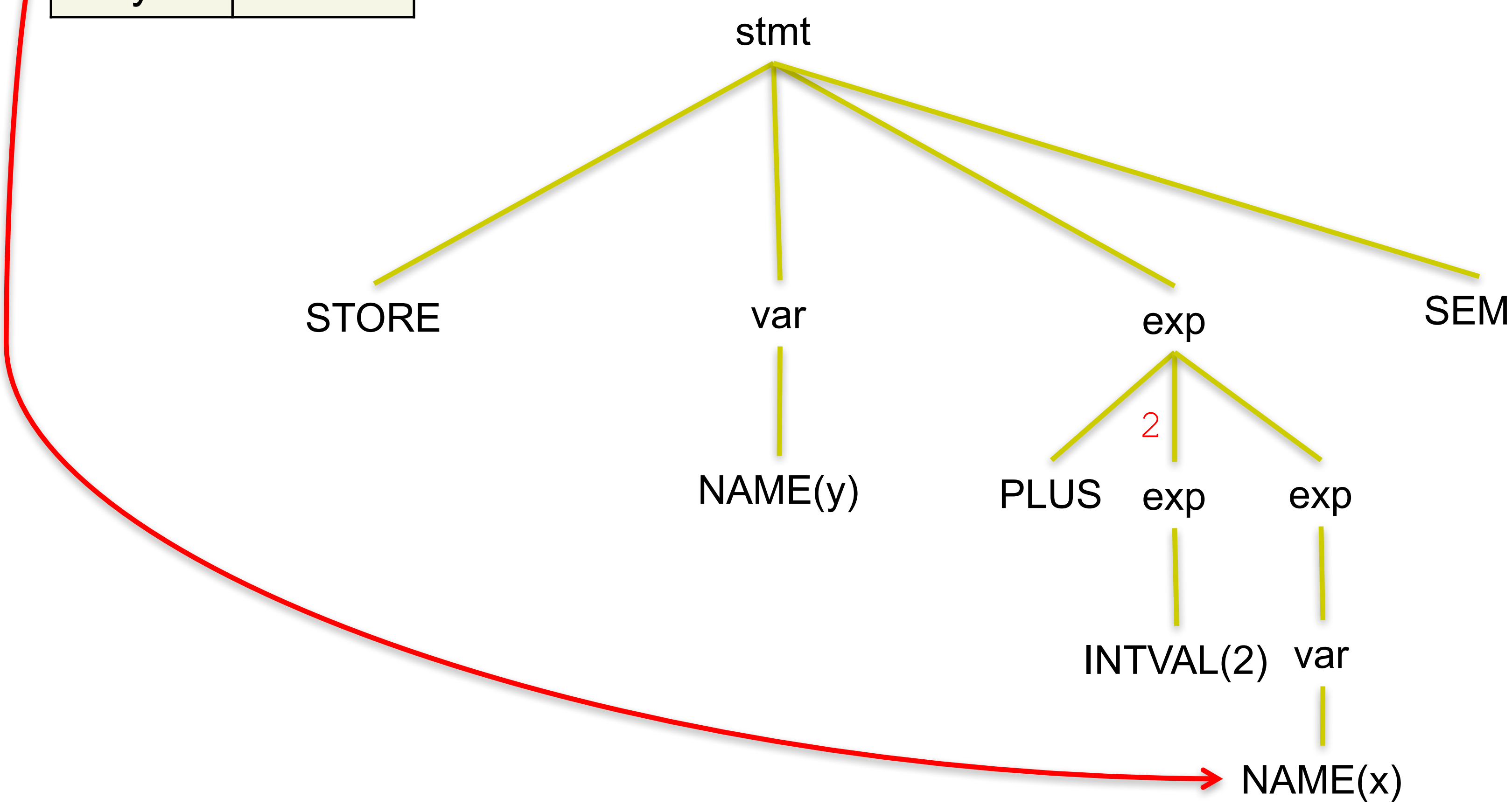
Symbol Table	
x	3
y	???

Action: interpret NAME



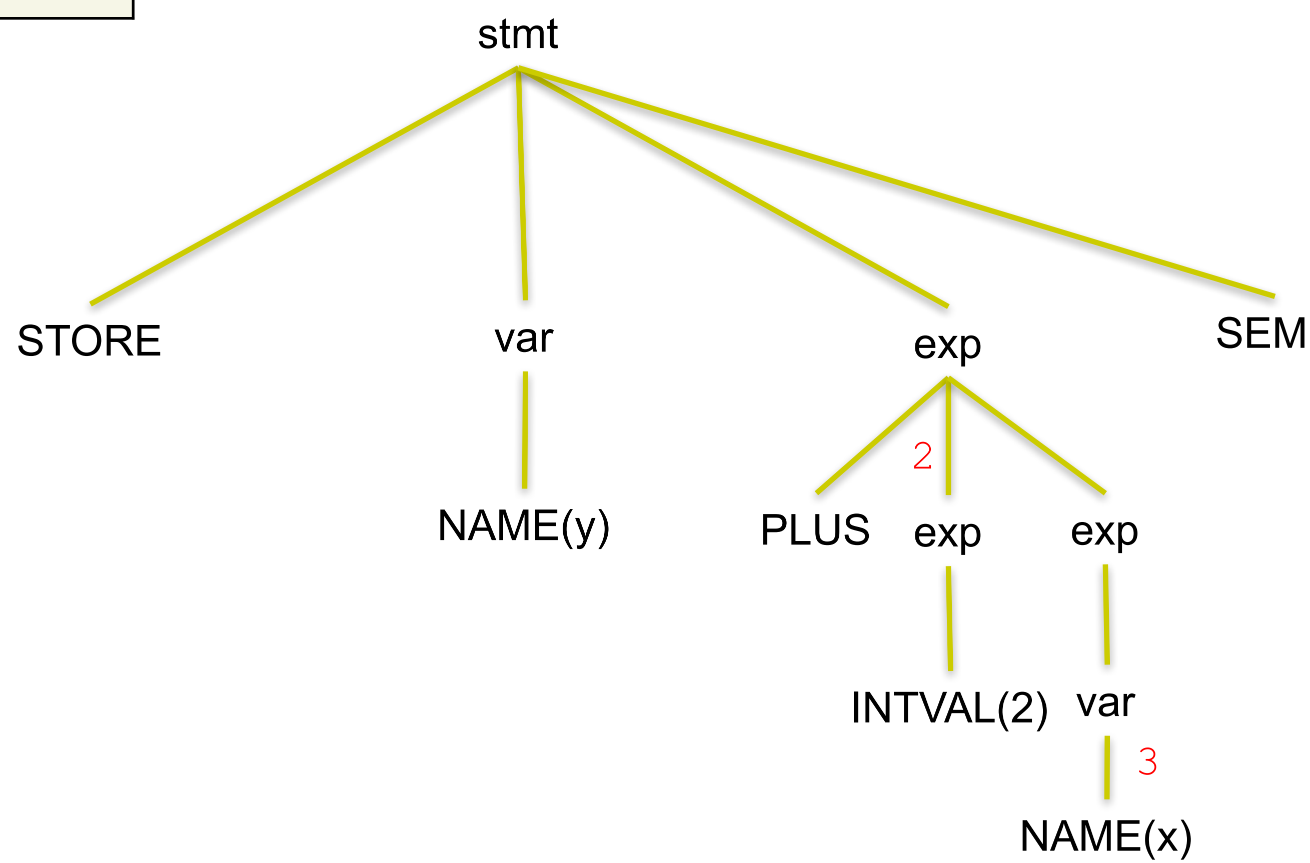
Symbol Table	
x	3
y	???

`Action: read symbol table`



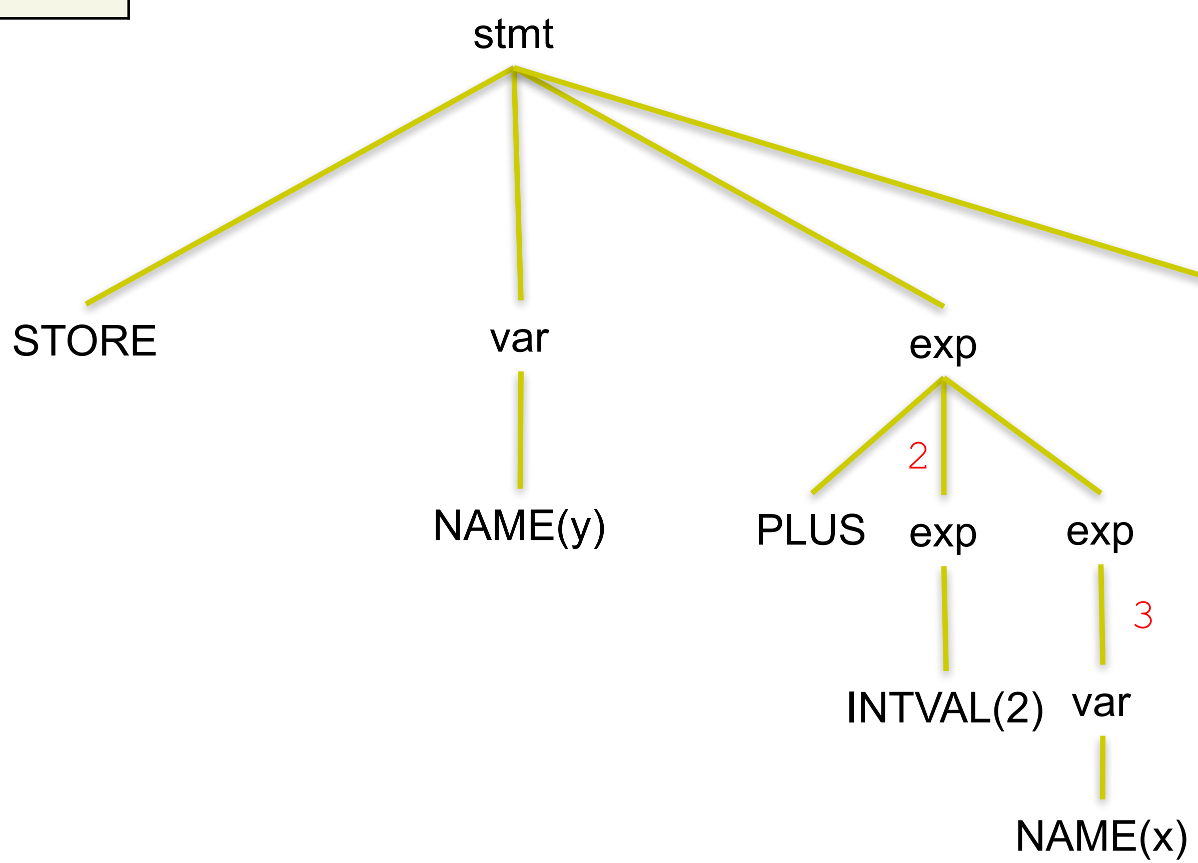
Symbol Table	
x	3
y	???

Action: propagate



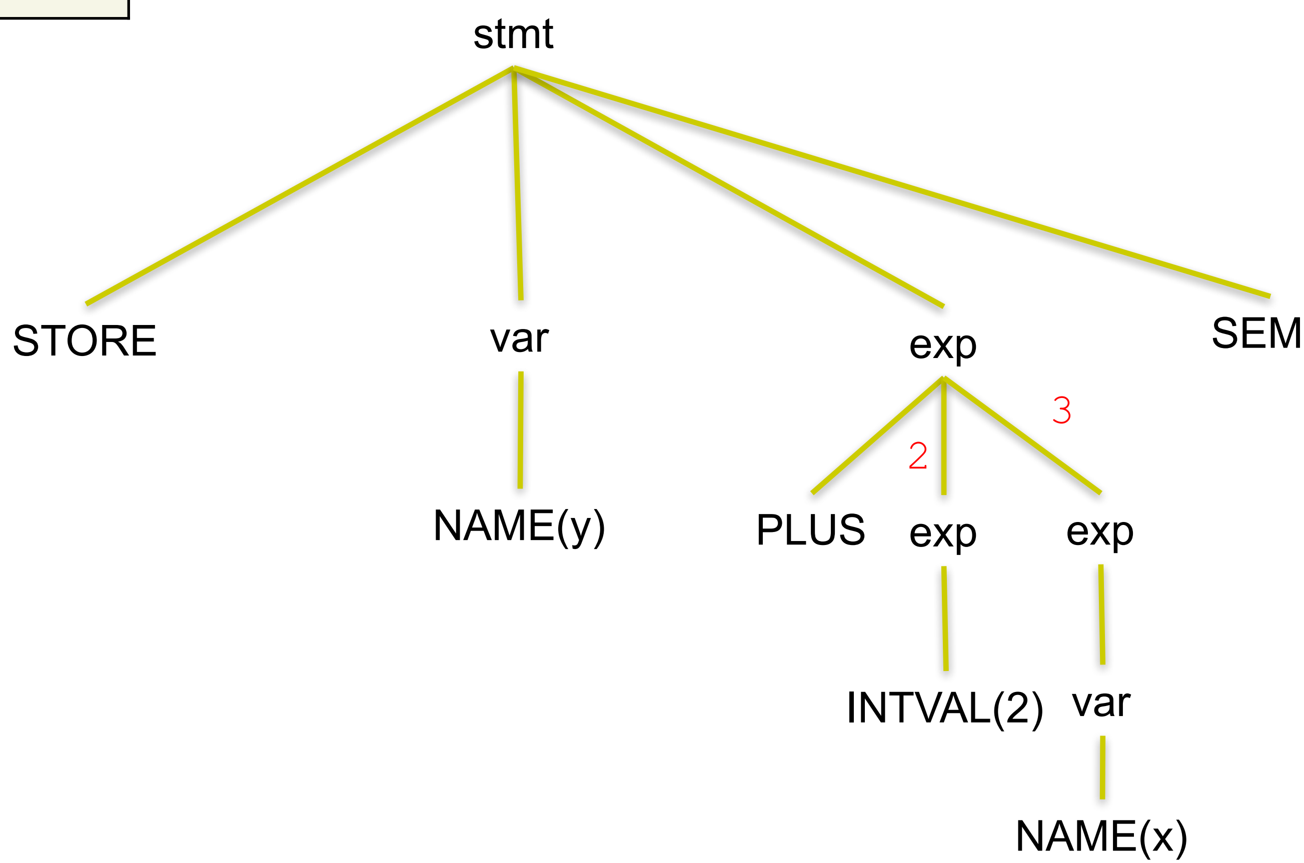
Symbol Table	
x	3
y	???

Action: propagate



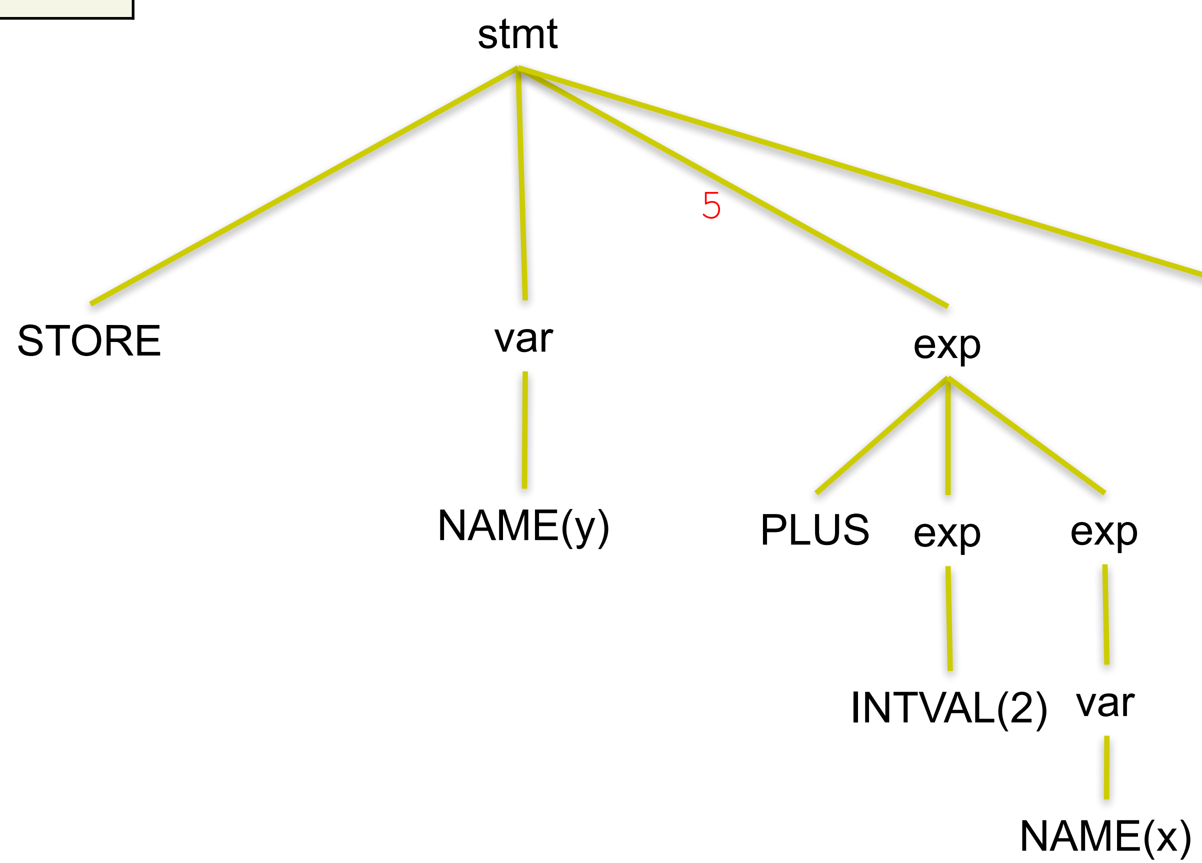
Symbol Table	
x	3
y	???

Action: add



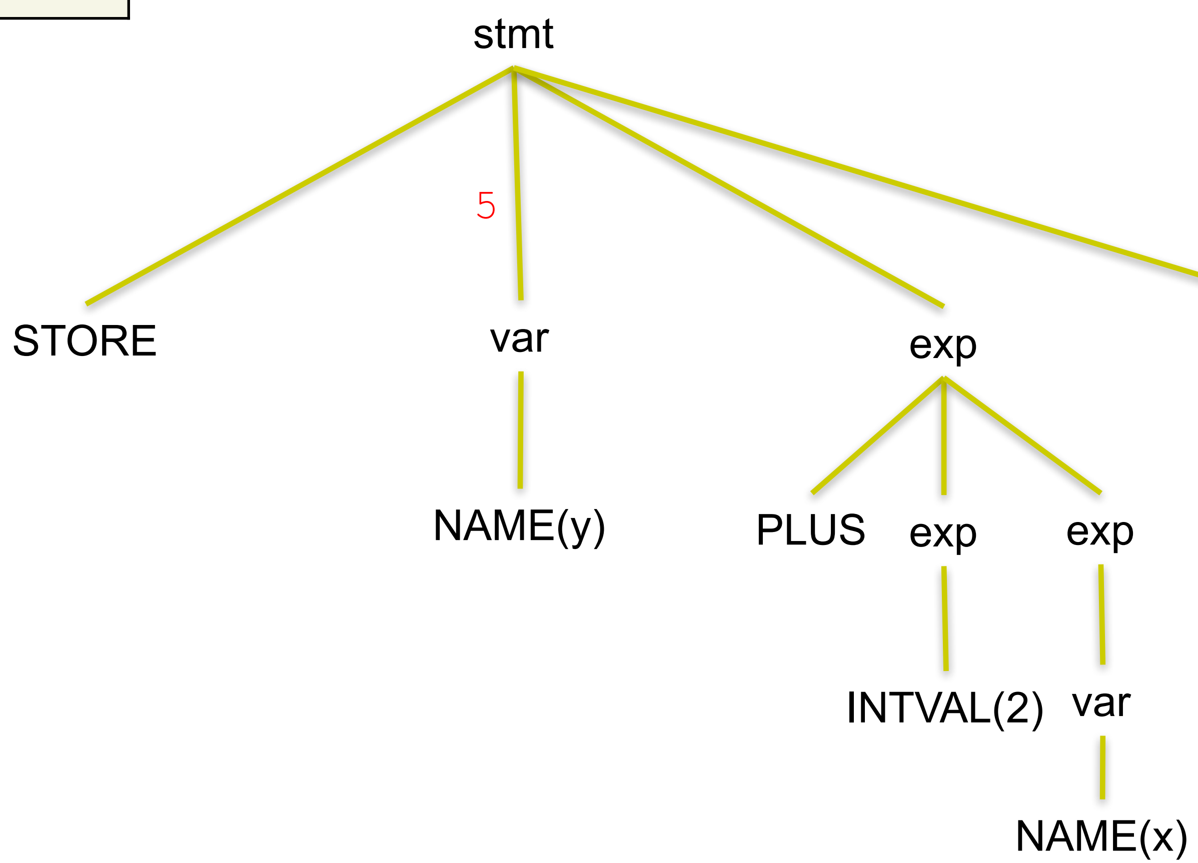
Symbol Table	
x	3
y	???

Action: propagate



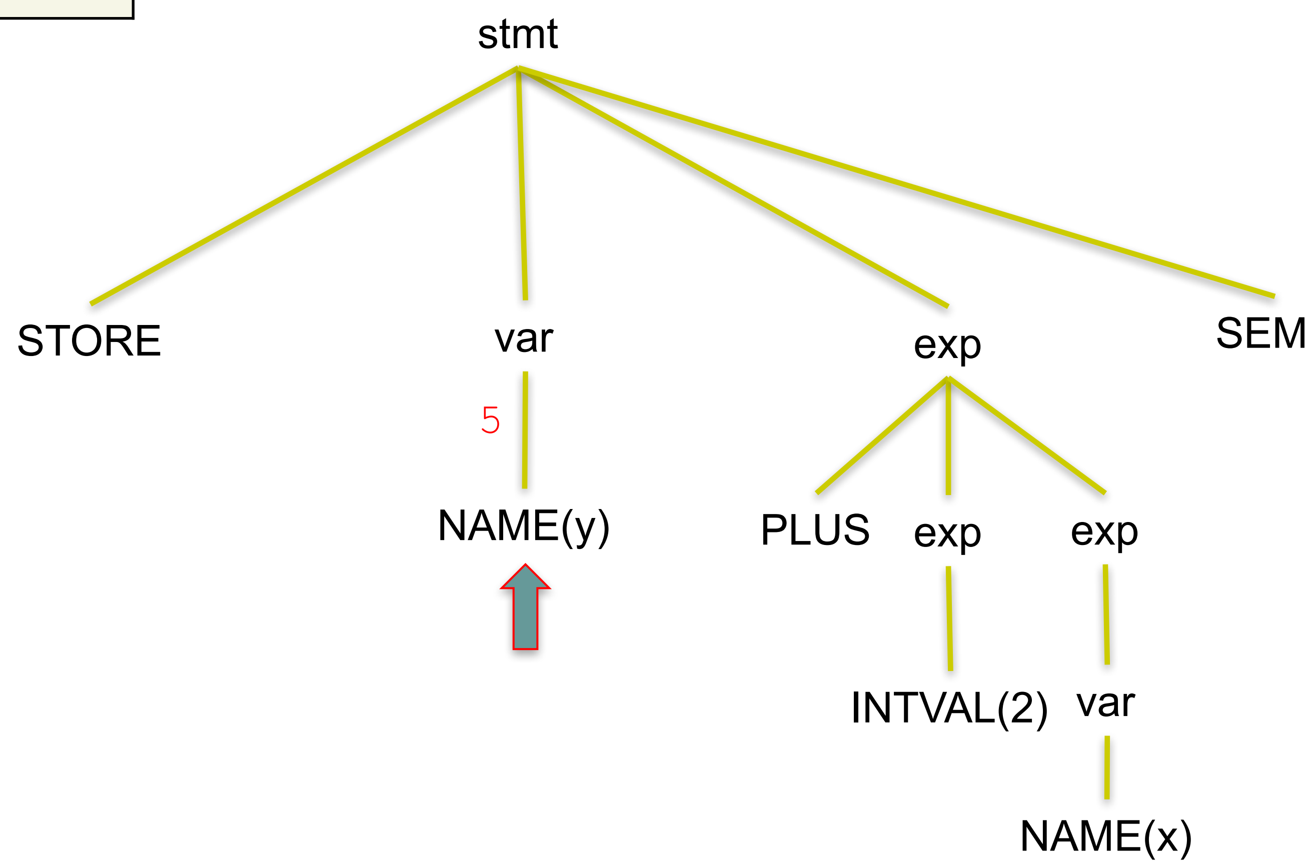
Symbol Table	
x	3
y	???

Action: propagate



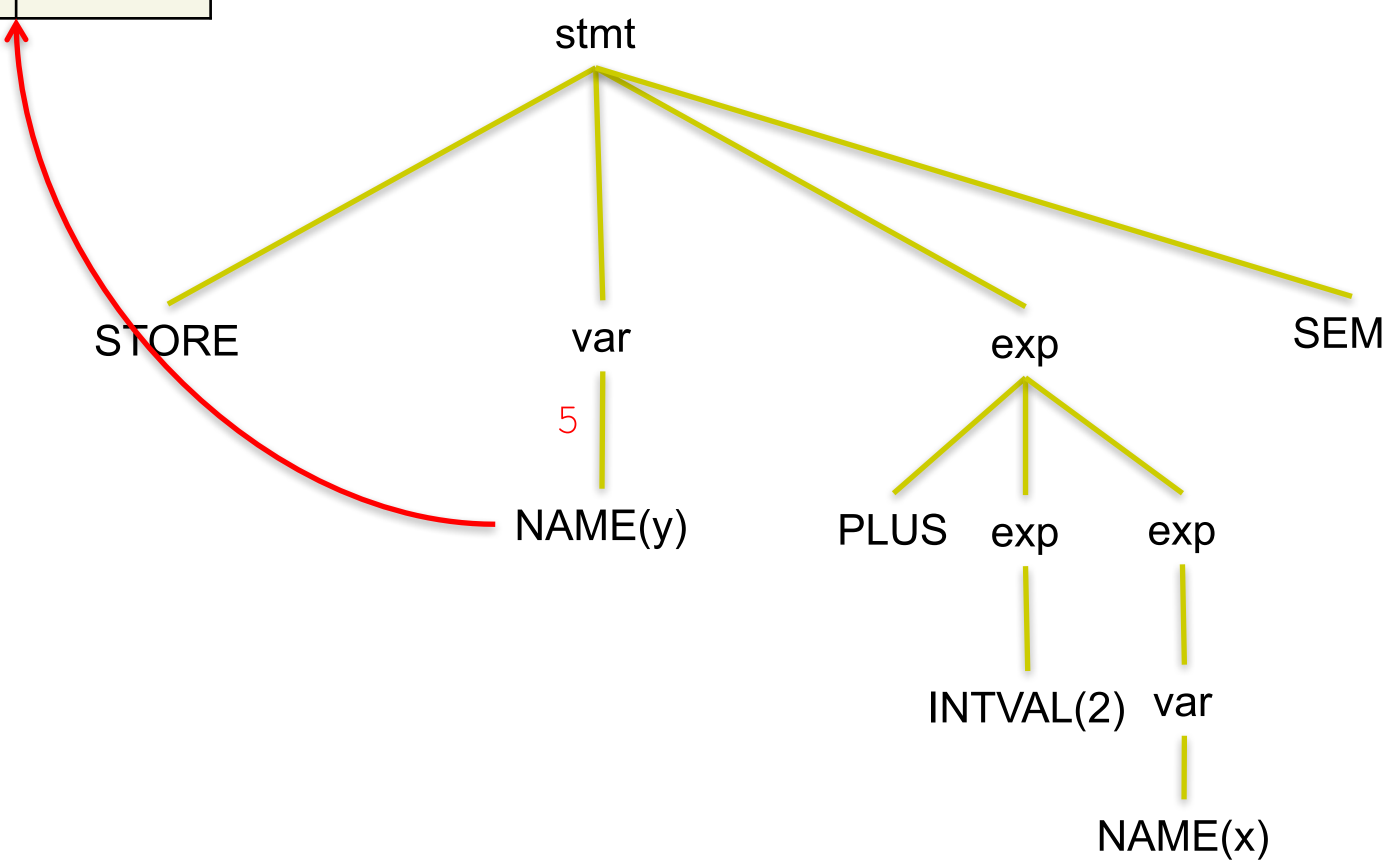
Symbol Table	
x	3
y	???

Action: interpret NAME



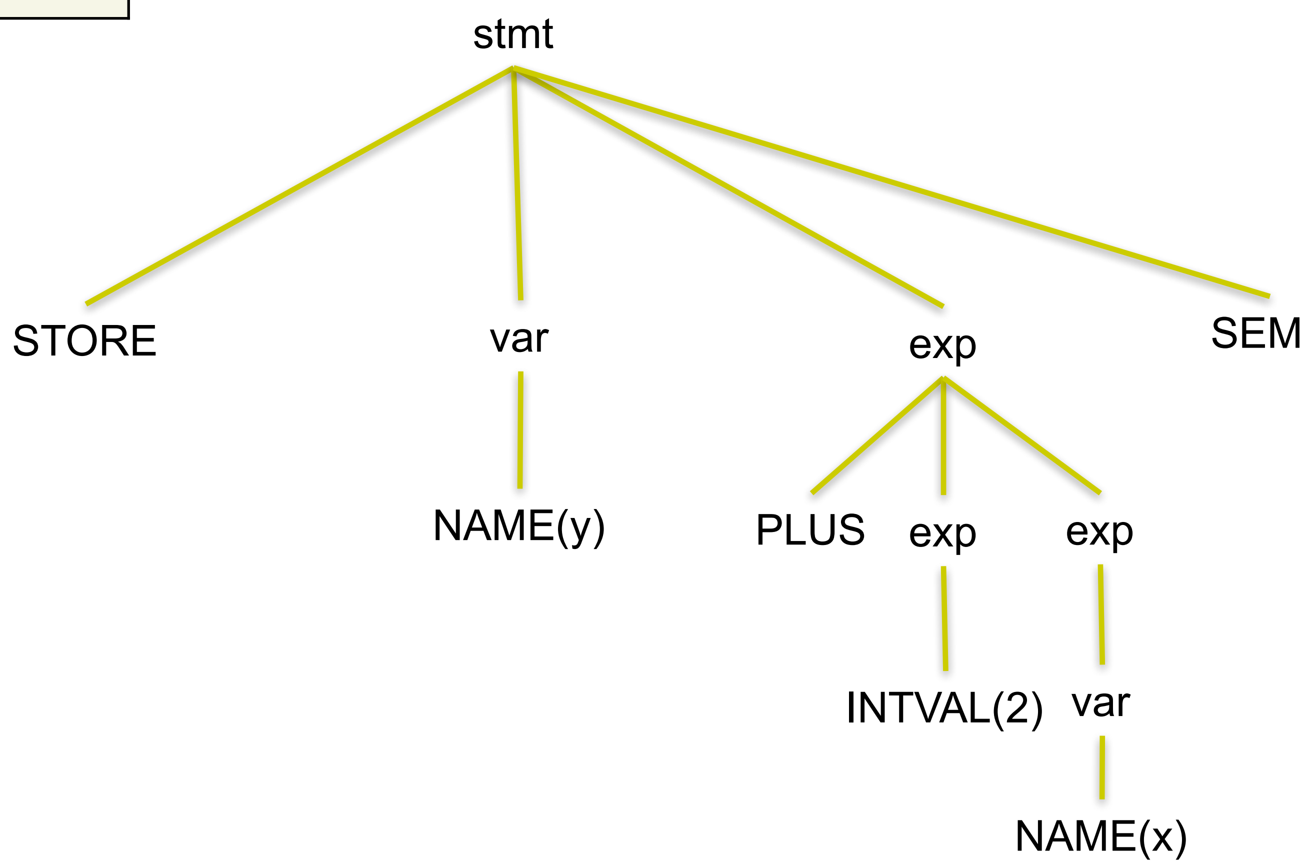
Symbol Table	
x	3
y	5

Action: write to symbol table



Symbol Table	
x	3
y	5

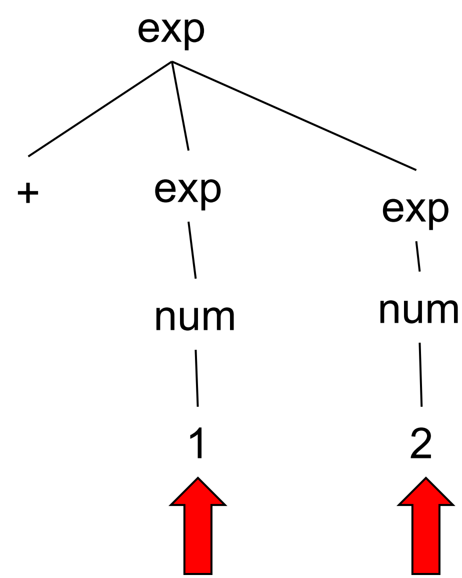
Action: done



Interpretation

- Consider the Exp1 expression: + 1 2

```
exp    :    '+' exp exp
       |    '-' exp exp
       |    '(' exp ')'
       |    var
       |    num
       ;
```



Interpretation means, computing the value of the root node.

We have to start at the leaves of the tree, that is where the primitive values are and proceed upwards…

What is the value at the root node?

Interpretation

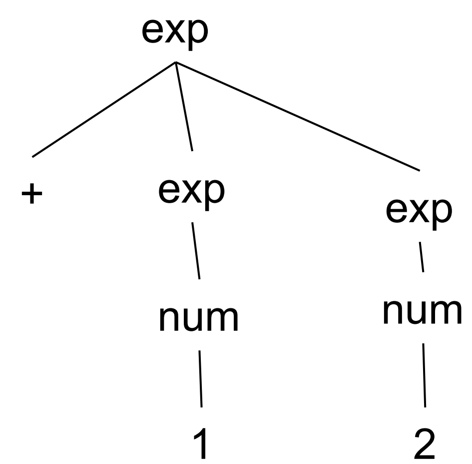
- We can rewrite the grammar to add the appropriate actions that have this bottom-up behavior.

```
exp     :     '+' exp exp
        ...
        |     num
        ;
```

```
def p_plus_exp(p):
    """
    exp : '+' exp exp
    """
    p[0] = p[2] + p[3]
```

```
def p_num_exp(p):
    "exp : num"
    p[0] = p[1]
```

```
def p_num(p):
    "num : NUMBER"
    p[0] = p[1]
```



Observation: the p list holds the values of all the symbols of the right side of a production. p[0] represents the value of the left side of the production:

```
exp : '+' exp exp
 0     1   2   3
```

Note: `p[1] == '+'`

Extended Exp1 Grammar

```
# %load code/exp1_lrinterp_gram.py
from ply import yacc
from exp1_lex import tokens, lexer

symbol_table = dict()

def p_prog(_):
    "prog : stmt_list"
  pass

def p_stmt_list(_):
  """
    stmt_list : stmt stmt_list
       | empty
  """
  pass

def p_print_stmt(p):
  "stmt : PRINT exp ';'"
  print("> {}".format(p[2]))

def p_store_stmt(p):
  "stmt : STORE NAME exp ';'"
  symbol_table[p[2]] = p[3]

…
```

Note: the lexer has not changed, only the grammar was extended with actions

```
…
def p_plus_exp(p):
  """
  exp : '+' exp exp
  """
  p[0] = p[2] + p[3]

def p_minus_exp(p):
  """
  exp : '-' exp exp
  """
  p[0] = p[2] - p[3]

def p_paren_exp(p):
  """
  exp : '(' exp ')'
  """
  p[0] = p[2]

def p_var_exp(p):
  "exp : var"
  p[0] = p[1]

def p_num_exp(p):
  "exp : num"
  p[0] = p[1]

def p_var(p):
  "var : NAME"
  p[0] = symbol_table.get(p[1], 0)

def p_num(p):
  "num : NUMBER"
  p[0] = p[1]

def p_empty(p):
    "empty :"
  pass

def p_error(t):
    print("Syntax error at '%s'" % t.value)

parser = yacc.yacc(debug=False, tabmodule='exp1parsetab')
```

Exp1 Lexer

```
# %load code/exp1_lex.py
# Lexer for Exp1

from ply import lex

reserved = {
  'store' : 'STORE',
  'print' : 'PRINT'
}

literals = [';','+','-','(',')']

tokens = ['NAME','NUMBER'] + list(reserved.values())

t_ignore = ' \t'

def t_NAME(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value,'NAME')    # Check for reserved words
    return t

def t_NUMBER(t):
    r'[0-9]+'
    t.value = int(t.value)
    return t

def t_NEWLINE(t):
  r'\n'
  pass

def t_error(t):
    raise SyntaxError("Illegal character {}".format(t.value[0]))

# build the lexer
lexer = lex.lex()
```

Putting this all together

- To finish the interpreter…
  - We have to create a top-level driving function that finds and connects the input file to the lexer/parser.

```
from exp1_lrinterp_gram import parser

def exp1_lrinterp(input_stream = None):
    'A driver for our LR Exp1 interpreter.'

    if not input_stream:
        input_stream = input("exp1 > ")

    parser.parse(input_stream)
```

Putting this all together

- We now have an interpreter that can run programs such as:

```
store y 3;
store x 2;
print + x y;
```

```
In [2]: from exp1_lrinterp import exp1_lrinterp

In [3]: program = \
        """
        store y 3;
        store x 2;
        print + x y;
        """

In [4]: exp1_lrinterp(program)

        > 5

In [ ]:
```

Reading

- Chapter 3
- Assignment #3 – please see website